Deductive Verification of Pipelined Machines Using First-Order Quantification*

Sandip Ray and Warren A. Hunt, Jr.

Department of Computer Sciences, University of Texas at Austin
{sandip,hunt}@cs.utexas.edu
http://www.cs.utexas.edu/users/{sandip,hunt}

Abstract. We outline a theorem-proving approach to verify pipelined machines. Pipelined machines are complicated to reason about since they involve simultaneous overlapped execution of different instructions. Nevertheless, we show that if the logic used is sufficiently expressive, then it is possible to relate the executions of the pipelined machine with the corresponding Instruction Set Architecture using (stuttering) simulation. Our methodology uses first-order quantification to define a predicate that relates pipeline states with ISA states and uses its *Skolem witness* for correspondence proofs. Our methodology can be used to reason about *generic* pipelines with interrupts, stalls, and exceptions, and we demonstrate its use in verifying pipelines mechanically in the ACL2 theorem prover.

1 Introduction

This paper is concerned with formal verification of pipelined machines. Pipelining is a key feature in the design of today's microprocessors. It improves performance by temporally overlapping execution of different instructions; that is, by initiating execution of a subsequent instruction before a preceding instruction has been completed. To formally verify modern microprocessors, it is imperative to be able to effectively reason about modern pipelines.

Formal verification of microprocessors normally involves showing some correspondence between a microarchitectural implementation (MA) and its corresponding Instruction Set Architecture (ISA). The ISA is typically a non-pipelined machine which executes each instruction atomically. For non-pipelined microprocessors, one typically shows *simulation correspondence* (Fig. 1): If an implementation state is externally equal to a specification state, then executing one instruction in both the microarchitecture and the ISA results in states that are externally equal [1, 2]. This implies that for every (infinite) execution of MA, there is a *corresponding* (infinite) execution of the ISA with the same visible behavior. However, for pipelined machines, such correlations are difficult to establish because of *latency* in the pipelines. As a result, a large number of correspondence notions have been used to reason about pipelined machines.

* Support for this work was provided in part by the SRC under contract 02-TJ-1032.

R. Alur and D.A. Peled (Eds.): CAV 2004, LNCS 3114, pp. 31–43, 2004.

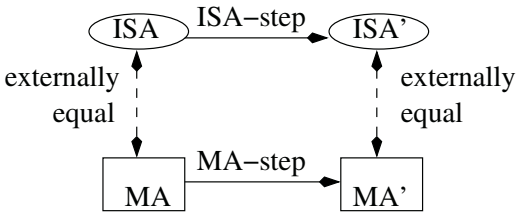

Fig. 1. Commutative Diagram for Simulation Correspondence

As features like interrupts, and out-of-order instruction execution have been modeled and reasoned about, notions of correctness have had to be modified and extended to account for these features. Consequently, the correspondence theorems have become complicated, difficult to understand, and even controversial [3]. Further, the lack of uniformity has made composition of proofs of different components of a modern processor cumbersome and difficult.

In this paper, we argue that in a logic that allows arbitrary first-order quantification and Skolemization, simulation-based correspondence is still effective and sufficient for reasoning about modern pipelines. Our notion of correctness is *stuttering simulations* [2], and preserves both *safety* and *liveness* properties. The chief contribution of this work is to show how to effectively use quantification to define a correspondence relating the states of a pipelined machine with those of its ISA. Our work makes use of the Burch and Dill *pipeline flushing diagram* [4] (Fig. 2), to derive the correspondence without complicated invariant definitions or characterization of the precise timing between the execution of different instructions in the pipeline. Indeed, our techniques are generic and we are able to apply the *same* notion of correspondence to pipelines with stalls, interrupts, exceptions, and out-of-order execution, and verify such machines mechanically with reasonable automation.

In this section, we first survey related approaches and notions of correctness used in verification of pipelined machines. We then describe our approach and proof methodology in greater detail.

1.1 Related Work

Reasoning about pipelines has been an active area of research. Aagaard *et al.* [3] provides an excellent survey and comparison of the different techniques. Some of the early studies have used *skewed abstraction functions* [5, 6] to map the states of the pipeline at different moments to a single ISA state. The skewed abstraction functions need to precisely characterize all timing delays and hence their definitions are both complex and vulnerable to design modifications. By far the most popular notion of correctness used has been the Burch and Dill *pipeline flushing diagram* [4] (Fig. 2). The approach is to use the implementation itself to flush the pipeline by not permitting new instructions to be fetched, and projecting the programmer-visible components of the flushed state to de-

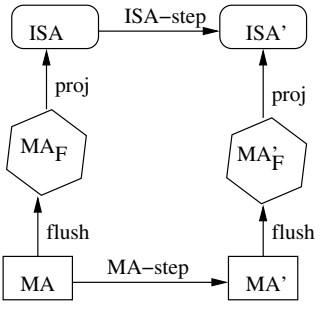

Fig. 2. Burch and Dill Pipeline Flushing Diagram

fine the corresponding ISA state. Sawada and Hunt [7, 8] use a variant called *flush point alignment* to verify a complicated pipelined microprocessor with exceptions, stalls, and interrupts using the ACL2 theorem prover. They use an intermediate data structure called MAETT to keep track of the history of execution of the different instructions through the pipeline. Hosabettu *et al.* [9] use another variant, *flush point refinement*, and they verify a deterministic out-of-order machine using *completion functions*. Both of these approaches require construction of complicated invariants to demonstrate correlation between the pipelined machine and the ISA. In addition, there have also been compositional model checking approaches to reason about pipelines. For example, Jhala and McMillan [10] use symmetry, temporal case-splitting, and data-type reduction to verify out-of-order pipelines. While more automatic than theorem proving, applicability of the method in practice requires the user to explicitly decompose the proof into manageable pieces to alleviate *state explosion*. Further, it relies on symmetry assumptions which are often violated by the heterogeneity of modern pipelines. Finally, there has been work on using a combination of decision procedures and theorem proving to verify modern pipelines [11, 12], whereby decision procedures have been used to assist the theorem prover in verifying state invariants.

Manolios [13] shows logical problems with the Burch and Dill notion and provides a general notion of correctness using *well-founded equivalence bisimulation* (WEB) [14]. He uses it to reason about several variants of Sawada's pipelines [7]. This, to our knowledge, is the first attempt in reasoning about pipelines using a general-purpose correctness notion expressing both safety and liveness properties. Our notion of correctness and proof rules are a direct consequence of the work with WEBs. The basic difference between our approach and that of Manolios is in the methodology used for relating pipeline states with ISA states. Since this methodology is a key contribution of this paper, we compare our approach with his in greater detail after outlining our techniques in Section 3.

1.2 Overview of Our Approach

We now describe how it is possible to relate pipeline states with the corresponding ISA using simulation-based correspondence in spite of overlapped execution

of different instructions. Note that our quest here is for a *generic* approach rather than the verification of one specific design. To convey the basic idea we present a simplified view of pipelined machines, namely in-order execution and completion of at most one instruction in a clock cycle. A precise mathematical description is presented later in Section 3, and features like interrupts, out-of-order execution etc., are dealt with in Section 4.

Consider the pipelined machine in some state MA, where a specific instruction i_1 is poised to complete. Presumably, then, before reaching MA, the machine must have encountered some state MA_1 in which i_1 was poised to enter the pipeline. Call MA_1 the *witnessing state* of MA. Assuming that instructions are completed in order, all and only the incomplete instructions at state MA_1 (meaning, all instructions before i_1) must complete before the pipeline can reach state MA starting from MA_1. Now consider the state MA_{1F} obtained by flushing MA_1. The flushing operation also completes all incomplete instructions without fetching any new instructions; that is, MA_{1F} has all instructions before i_1 completed, and is poised to fetch i_1. If only completed instructions affect the visible behavior of the machine, then this suggests that MA_{1F} and MA must be externally equal[1].

Based on the above intuition, we define a relation *sim* to correlate pipeline states with ISA states: MA is related to ISA if and only if there exists a witnessing state MA_1 such that ISA is the projection of the visible components of MA_{1F}. Recall that projection preserves the visible components of a state. From the arguments above, whenever states MA and ISA are related by *sim* they are externally equal. Thus to establish simulation correspondence (Fig. 1), we need to show that if MA is related to ISA, and MA' and ISA' are states obtained by 1-step execution from MA and ISA respectively, then MA' and ISA' are related by *sim*. Our approach is shown in Fig. 3. Roughly, we show that if MA and ISA

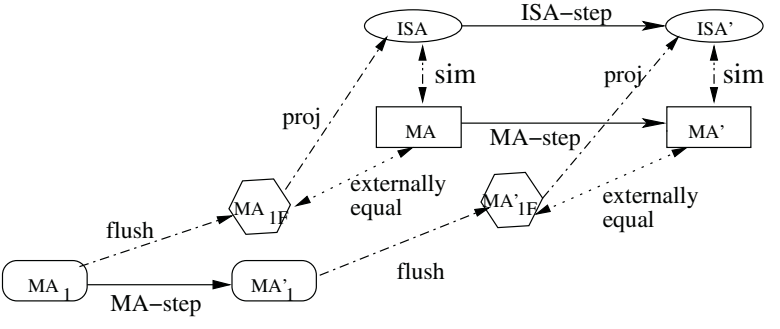

Fig. 3. Use of Burch and Dill and Symbolic Execution to obtain Simulation-based refinement proofs

[1] Notice that MA and MA_{1F} would possibly have different values of the program counter. This is normally addressed [13] by excluding the program counter from the observations (or *labels*) of a state.

are related, and MA_1 is the corresponding witnessing state for MA, then one can construct a corresponding witnessing state for MA' by running the pipeline for a sufficient number of steps from MA_1. In particular, ignoring the possibility of stalls (or bubbles) in the pipeline, the state following MA_1, which is MA'_1, is a witnessing state of MA', by the following argument. Execution of the pipeline for one cycle from MA completes i_1 and the next instruction, i_2, *after* i_1 in program order is poised to complete in state MA'. The witnessing state for MA' thus should be poised to initiate i_2 in the pipeline. And indeed, execution for one cycle from MA_1 leaves the machine in a state in which i_2 is poised to enter the pipeline. (We make this argument more precise in the context of an example pipeline using Lemma 2 in Section 3.) Finally, the correspondence between MA'_1 and ISA' follows by noting that the MA states MA_1, MA'_1, and ISA states ISA and ISA' satisfy the Burch and Dill *pipeline flushing diagram*.

The reader should note that the above proof approach depends on our determining the witnessing state MA_1 given MA. However, since MA_1 is a state that occurs in the "past" of MA, MA might not retain sufficient information for *computing* MA_1. In general, to compute witnessing states, one needs to define an intermediate machine to keep track of the history of execution of the different instructions in the pipeline. However, an observation of this paper is that given a sufficiently expressive logic, the witnessing state need not be constructively computed. Rather, we can simply define a predicate specifying "some witnessing state exists". Skolemization of the predicate then produces a witnessing state.

In the presence of stalls, a state MA might not have *any* instruction poised to complete. Even for pipelines without stalls, no instruction completes for several cycles after the initiation of the machine. Correspondence in the presence of such bubbles is achieved by allowing finite stutter in our verification framework. In other words, if MA is related to ISA, and MA has no instruction to complete, then MA' is related to ISA instead of ISA'. Our proof rules guarantee such stuttering is finite, allowing us to preserve both safety and liveness properties. Note that correspondence frameworks allowing a finite stutter are known to be effective in relating two system models at different levels of abstraction using simulation and bisimulation. In ACL2, stuttering has been used with simulation and bisimulation to verify concurrent protocols [15, 14], and indeed, pipelined machines [13].

The remainder of the paper is organized as follows. We discuss our notion of correspondence and the associated proof rules in Section 2. In Section 3, we discuss the correspondence proof for an example pipelined machine. In Section 4, we show how our method can be used to reason about pipelines with interrupts, out-of-order execution etc. Finally, we conclude in Section 5. All the theorems described here have been proven with the ACL2 theorem prover [16, 17]. ACL2 is an essentially first-order logic of total recursive functions, with induction up to ϵ_0 and support for first-order quantification via Skolemization. However, this paper does not assume any prior exposure to ACL2; we use traditional mathematical notations for our presentation.

2 Refinements

Our proof rules relate infinite executions of two computing systems using single-step theorems. The rules are adapted from the proof rules for *stuttering simulation* [2], and a direct consequence of work with WEBs [14].

Computing systems are typically modeled in ACL2 as state machines by three functions namely *init*, *step*, and *label*, with the following semantics:

- The constant function $init()$ returns the initial state of the system.
- Given a state s and an external input i, the function $step(s, i)$ returns the state of the system after one clock cycle.
- For a state s, the function $label(s)$ returns valuations of the observable components of s [2].

Such models are common in ACL2 and have been found useful in verification of several hardware and software systems [18, 1]. Also, since ACL2 is a logic of total functions, the functions *step* and *label* above are total.

We now describe the proof rules. Given models $impl = \langle init_I, step_I, label_I \rangle$, and $spec = \langle init_S, step_S, label_S \rangle$, the idea is to define binary predicates *sim* and *commit* with the following properties:

1. $sim(init_I(), init_S())$,
2. $\forall p_I, p_S : sim(p_I, p_S) \Rightarrow label_I(p_I) = label_S(p_s)$,
3. $\forall p_I, p_S, i, \exists j : sim(p_I, p_S) \wedge commit(p_I, i) \Rightarrow sim(step_I(p_I, i), step_S(p_S, j))$,
4. $\forall p_I, p_S, i : sim(p_I, p_S) \wedge \neg commit(p_I, i) \Rightarrow sim(step_I(p_I, i), p_S)$.

Informally, *sim* relates the states of *impl* with the states of *spec*, and *commit* determines whether the "current step" is a stuttering step for *spec*. The above four conditions guarantee that for every (infinite) execution of *impl*, there exists a corresponding (infinite) execution of *spec* that has the same observations up to stuttering. To guarantee that stuttering is finite, we define a unary function *rank* with the following properties:

5. $\forall p : rank(p) \in W$, where W is a set known to be well-founded according some ordering relation $\prec$.
6. $\forall p_I, p_S, i : sim(p_S, p_I) \wedge \neg commit(p_I, i) \Rightarrow rank(step_I(p_I, i)) \prec rank(p_I)$.

We say that *impl* is a (stuttering) *refinement* of *spec*, denoted ($impl >> spec$), if there exist functions *sim*, *commit*, and *rank* that satisfy the six conditions above. The proof rules essentially guarantee that *impl* is a simulation of *spec* up to finite stuttering. The rules are analogous to proof rules for *stuttering simulation* [2] with the difference that our rules allow only one-sided stutter. We have not yet found a problem in which two-sided stutter has been necessary. Notice also that the notion of refinements is transitive, and this allows us to hierarchically compose proofs of different components of large practical systems.

Our description above shows how to relate two non-deterministic system models. A computing system $M = \langle init, step, label \rangle$ is *deterministic* if *step* is a

[2] The *label* is analogous to the valuation of atomic propositions in a Kripke Structure.

function of only the first argument, namely the current state. If *impl* is deterministic, then *commit* is a function of the current state alone. In addition, if *spec* is deterministic, then the condition 3 is modified to eliminate the choice of input for the next step of *spec*:

3. $\forall p_I, p_s, : sim(p_I, p_S) \wedge commit(p_I) \Rightarrow sim(step_I(p_I), step_S(p_S))$.

3 Pipeline Verification

We now show how to reason about pipelines using the correctness notion described in Section 2. For clarity of presentation, we consider a deterministic 5-stage example pipeline with in-order execution. Features like arbitrary stalls, interrupts, and out-of-order execution are dealt with in Section 4.

Our pipeline (Fig. 4) has *fetch*, *decode*, *operand-fetch*, *execute*, and *write-back* stages. The machine has a decoder, an ALU, a register file, and 4 latches

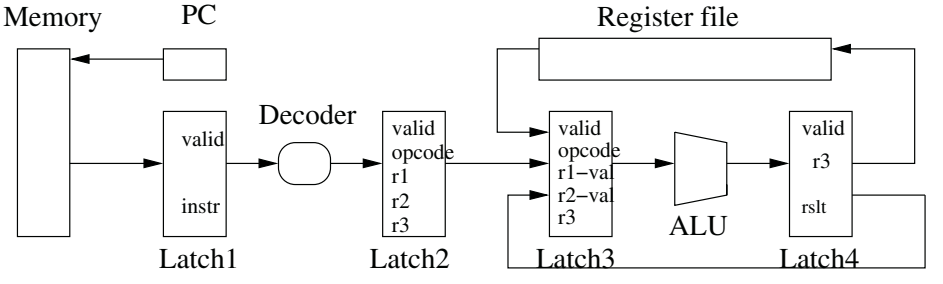

Fig. 4. A simple 5-stage pipeline

to store intermediate computations. Instructions are fetched from the memory location pointed to by the program counter (PC) and loaded into the first latch. The instructions then proceed in sequence through the different pipe stages in program order until they retire. Every latch has a *valid* bit to indicate if the latch is non-empty. We use a 3-address instruction format consisting of an opcode, two source registers, and a target register. While the machine allows only in-order execution, data forwarding is implemented from latch 4 to latch 3. Nevertheless, the pipeline can still have 1-cycle stall: If the instruction i_2 at latch 2 has as one of its source registers the target of the instruction i_3 at latch 3, then i_2 cannot proceed to latch 3 in the next state when i_3 completes its ALU operation.

The executions of the pipeline above can be defined as a *deterministic* model $\mathrm{MA} = \langle init_I, step_I, label_I \rangle$. We assume that $init_I$ has *some* program loaded in the memory, and *some* initial configuration of the register file, but an empty pipeline. Since we are interested in the updates of the register file, we let $label_I$ preserve the register file of a state. Finally, the ISA machine $\mathrm{ISA} = \langle init_S, step_S, label_S \rangle$ is merely one that executes the instruction pointed to by PC atomically; that is, it

fetches the correct operands, applies the corresponding ALU operation, updates the register file, and increments PC in one atomic step.

Notice that we have left the actual instructions unspecified. Our proof approach does not require complete specification of the instructions but merely the constraint that the ISA performs analogous atomic update for each instruction. In ACL2, we use constrained functions for specifying the updates applied to instructions at every stage of the pipeline.

Our goal now is to show that (MA >> ISA). We define *commit* so that *commit*(*MA*) is true if and only if latch 4 is *valid* in *MA*. Notice that whenever an MA state *MA* satisfies *commit*, some instruction is completed at the *next step*. It will be convenient to define functions characterizing partial updates of an instruction in the pipeline. Consequently, we define four functions run_1, run_2, run_3, and run_4, where $run_i(MA, inst)$ "runs" instruction *inst* for the first i stages of the pipe and updates the i-th latch. For example, $run_2(MA, inst)$ updates latch 2 with the decoded value of the instruction. In addition, we define two functions, *flush* and *stalled*. Given a pipeline state *MA*, *flush*(*MA*) returns the flushed pipeline state MA_F, by executing the machine for sufficient number of cycles without fetching any new instruction. The predicate *stalled* is true of a state *MA* if and only if both latches 2 and 3 are *valid* in *MA*, and the destination for the instruction at latch 3 is one of the sources of the instruction at latch 2. Notice that executing a pipeline from a *stalled* state does not allow a new instruction to enter the pipeline. Finally the function *proj*(*MA*) projects the PC, memory, and register file of *MA* to the ISA.

Lemma 1 is the Burch and Dill correctness for pipelines with stalls and was proved automatically by ACL2 using symbolic simulation.

Lemma 1. *For each pipeline state MA,*

$$proj(flush(step_I(MA))) = \begin{cases} step_S(proj(flush(MA))) & \text{if } \neg stalled(MA) \\ proj(flush(MA)) & \text{otherwise} \end{cases}$$

We now define *witnessing states*. Given states MA_1 and *MA*, MA_1 is a *witnessing state* of *MA*, recognized by the predicate *witnessing*(MA_1, *MA*), if (1) MA_1 is not *stalled*, and (2) *MA* can be derived from MA_1 by the following procedure:

1. *Flush* MA_1 to get the state MA_{1F}.
2. Apply the following update to MA_{1F} for $i = 4, 3, 2, 1$:
   - If latch i of *MA* is *valid* then apply run_i with the instruction pointed to by PC in MA_{1F}, correspondingly update latch i of MA_{1F}, and advance PC; otherwise do nothing.

We now define predicate *sim* as follows:

- $sim(MA, ISA) \doteq (\exists MA_1 : (witnessing(MA_1, MA)) \wedge (ISA = proj(flush(MA_1))))$

Now we show how the predicate *sim* can be used to show the conditions 1-6 in Section 2. First, the function *rank* satisfying conditions 5 and 6 can be defined by

simply noticing that for any state MA, *some* instruction always advances. Hence if i is the maximum number in MA such that latch i is *valid*, then the quantity $(5-i)$ always returns a natural number (and hence a member of a well-founded set) that decreases at every step. (We take i to be 0 if MA has no *valid* latch.) Also, $\mathit{witnessing}(\mathit{init}_I(), \mathit{init}_I())$ holds, implying condition 1. Further, condition 2 is trivial from definition of *witnessing*. We therefore focus here on only conditions 3 and 4. We first consider the following lemma:

Lemma 2. *For all MA, MA_1 such that $\mathit{witnessing}(MA_1, MA)$:*

1. $\mathit{witnessing}(MA_1, \mathit{step}_I(MA))$ if $\neg \mathit{commit}(MA)$
2. $\mathit{witnessing}(\mathit{step}_I(\mathit{step}_I(MA_1)), \mathit{step}_I(MA))$ if $\mathit{commit}(MA)$ $\wedge \mathit{stalled}(\mathit{step}_I(MA_1))$
3. $\mathit{witnessing}(\mathit{step}_I(MA_1), \mathit{step}_I(MA))$ if $\mathit{commit}(MA) \wedge \neg \mathit{stalled}(\mathit{step}_I(MA_1))$

The lemma merely states that if MA is not a *commit* state, then stepping from MA preserves the witnessing state, and otherwise the witnessing state for $\mathit{step}_I(MA)$ is given by the "next non-stalled state" after MA_1. The lemma can be proved by symbolic execution of the pipelined machine. We can now prove the main technical lemma that guarantees conditions 3 and 4 of Section 2.

Lemma 3. *For all MA and ISA, such that $\mathit{sim}(MA, ISA)$:*

1. $\mathit{sim}(\mathit{step}_I(MA), ISA)$ *if* $\neg \mathit{commit}(MA)$.
2. $\mathit{sim}(\mathit{step}_I(MA), \mathit{step}_S(ISA))$ *if* $\mathit{commit}(MA)$.

Proof. Let MA_1 be the Skolem witness of $\mathit{sim}(MA, ISA)$. Case 1 follows from Lemma 2, since $\mathit{witnessing}(MA_1, \mathit{step}_I(MA))$ holds. For case 2, we consider only the situation $\neg \mathit{stalled}(\mathit{step}_I(MA_1))$ since the other situation is analogous. But by Lemma 1 and definition of *witnessing*, $\mathit{proj}(\mathit{flush}(\mathit{step}_I(MA_1))) = \mathit{step}_S(ISA)$. The result now follows from Lemma 2. □

We end this section with a brief comparison between our approach and that of Manolios [13]. We focus on a comparison with this work since unlike other related work, this approach uses a uniform notion of correctness that is applicable to both pipelined and non-pipelined microarchitectures. Indeed our use of stuttering simulation is a direct consequence of this work. The basic difference is in the techniques used to define the correspondence relation to relate MA states with ISA states. While we define a quantified predicate to posit the existence of a witnessing state, Manolios defines a refinement map from the states of MA to states of ISA as follows: Point the PC to the next instruction to complete and invalidate all the instructions in the pipe[3]. Notice immediately that the method requires that we have to keep track of the PC of each intermediate instruction. Also, as the different pipeline stages update the instruction, one needs

[3] Manolios also describes a "flushing proof" and shows that flushing can relate inconsistent MA states to ISA states. But our application of flush is different from his in that we flush MA_1 rather than the current state MA. Our approach has no such problems.

some invariant specifying the transformations on each instruction at each stage. Short of computing such invariants based on the structure of the pipeline and the functionality of the different instructions, we believe that any generic approach to determine invariants will reduce his approach to ours, namely defining a quantified predicate to posit the existence of a witnessing state.

4 Generalization

Stuttering simulation can be used to verify pipelines with advanced features. In this section, we consider arbitrary but finite stalls, interrupts, and out-of-order instruction execution. Stuttering simulation cannot be used directly for machines with out-of-order completion and multiple instruction completion. In Section 4.4 we discuss an approach to circumvent such limitations.

4.1 Stalls

The pipeline in Fig. 4 allowed single-cycle stalls. Pipelines in practice can have stalls ranging over multiple cycles. If the stall is finite, it is easy to use stuttering simulation to reason about such pipelines. Stalls affect Lemma 2 since given a witnessing state MA_1 of MA, the witnessing state for $step_I(MA)$ is given by the "next non-stalled state" after MA_1. But such a state is given by $clock(step_I(MA_1))$, where the function *clock* (defined below) merely counts the number of steps to reach the first non-stalled state. Finiteness of stalls guarantees that the recursion terminates.

$$clock(s) \doteq \begin{cases} 0 & \text{if } \neg stalled(s) \\ 1 + clock(step_I(s)) & \text{otherwise} \end{cases}$$

4.2 Interrupts

Modern pipelines allow interrupts and exceptions. To effectively reason about interrupts, we model both MA and ISA as non-deterministic machines, where the "input parameter" is used to decide whether the machine is to be interrupted at the current step or proceeds with normal execution. Recall that our notion of correspondence can relate non-deterministic machines. Servicing the interrupt might involve an update of the visible components of the state. In the ISA, we assume that the interrupt is serviced in one atomic step, while it might take several cycles in MA.

We specify the witnessing states for an interruptible MA state as follows: MA_1 is a witnessing state of MA if either (i) MA is not within any interrupt and MA_1 initiates the instruction next to be completed in MA, or (ii) MA is within some interrupt and MA_1 initates the corresponding interrupt. Then *commit* is true if either the current step returns from some interrupt service or completes an instruction. Assuming that pipeline executions are not interleaved with the interrupt processing, we can then show (MA >> ISA) for such non-deterministic

machines. We should note here that we have not analyzed machines with nested interrupts yet. But we believe that the methodology can be extended for nested interrupts by the witnessing state specifying the initiation of the most recent interrupt in the nest.

4.3 Out-of-Order Execution

Our methodology can handle pipelines with out-of-order instruction execution as long as the instructions are initiated to the pipeline and completed in program order. For a pipeline state MA, we determine the instruction i_1 that is next to be completed, by merely simulating the machine forward from MA. We then specify MA_1 to be a witnessing state of MA if i_1 is initiated into the pipeline at MA_1. Notice that since instructions are initiated and completed in-order, any instruction before i_1 in program order must have been already initiated in the pipeline in state MA_1. Since flushing merely involves executing the pipeline without initiating any instruction, flushing from state MA_1 will therefore produce the state MA_{1F} with the same visible behavior as state MA.

4.4 Out-of-Order and Multiple Instruction Completion

Modern pipelines allow completion of multiple instructions at the same clock cycle, and out-of-order completion of instructions. Such features cannot be directly handled by our approach. In this section, we outline the problem and discuss a possble approach. We admit, however, that we have not attempted to verify pipelines with these features and our comments are merely speculative.

Consider a pipeline state MA poised to complete two instructions i_1 and i_2 at the same cycle. Assume that i_1 updates register r_1 and i_2 updates register r_2. Thus the visible behavior of the pipeline will show simultaneous updates of the two registers. The ISA, however, can only update one register at any clock cycle. Thus, there can be *no* ISA state ISA with the properties that (i) MA and ISA are externally equal, and (ii) executing both machines from MA and ISA results in states that are externally equal, even with possible stutter. In other words, there can be no simulation relation relating MA and ISA. The arguments are applicable to out-of-order completion as well, where the updates corresponding to the two instructions are "swapped" in the pipelined machine.

Since multiple and out-of-order completion affect the visible behavior of the pipelined machine, we need to modify the execution of the ISA to show correspondence. We propose the following approach: The ISA, instead of executing single instructions atomically, non-deterministically selects a *burst* of instructions. Each *burst* consists of a set of instruction sequences with instructions in *different* sequences not having any data dependency. The ISA then executes each sequence in a burst atomically, choosing the order of the sequences non-deterministically, and then selects the next burst after completing all sequences. Notice that our "original" ISA can be derived from such an ISA by letting the bursts be singleton sets of one instruction.

We are pursuing the "modified ISA" approach to verify practical pipelines. However, we have not verified any complicated machine using this approach, and investigation of the technique is a future area of research.

5 Summary and Conclusions

We have shown how to apply a uniform and well-studied notion of correspondence, namely stuttering simulation, to verify pipelined machines. Since the notion is compositional, proofs of pipelines can be directly composed with proofs of other components of a microprocessor. Further, the use of *witnessing states* enables us to merely "flush" the pipelined machine to define the simulation relation without constructing complicated structural invariants.

Our work demonstrates the importance of first-order quantification in verification of practical computing systems. We have seen how quantification enables us to specify the witnessing state non-constructively by using the Skolem witness of a quantified predicate. Since the witnessing state is a state that has been encountered in the "past", constructing the state would require keeping track of the history of execution via an intermediate machine, and definition of a complicated invariant on that machine. The importance of first-order quantification is often overlooked in the use of a general-purpose theorem prover, especially by the ACL2 community. While ACL2 allows arbitrary first-order quantification, ACL2 proofs typically use constructive recursive functions. In this context, it should be noted that a key advantage of the use of theorem-proving over algorithmic decision procedures like model-checking is the expressivity of the logic. Beyond this, theorem-proving normally involves more manual effort. Thus to successfully use theorem-proving with reasonable automation, it is important to make effective use of expressivity. We have found quantification to be particularly useful in many circumstances for "backward simulation", when the property preserved in a state is guaranteed by a state encountered by the machine in the past. Aside from defining witnessing states for pipelined machines, backward simulation is useful, for example, in specifying weakest preconditions for operationally modeled sequential programs. We are working on identifying other situations in which quantification can significantly reduce manual effort in deductive verification of modern computer systems.

We have demonstrated the application of our approach to verify pipelines with stalls, interrupts, and out-of-order execution using stuttering simulation. As mentioned in Section 4.4, we cannot yet handle out-of-order completion and multiple instruction completion. We plan to analyze the extention outlined in Section 4.4 to reason about more complicated, practical pipelines.

Acknowledgments

Rob Sumners was involved with the initial discussions about the use of quantification for reasoning about pipelines. Anna Slobodová read a draft of this paper and made several useful suggestions.

References

1. W. A. Hunt, Jr.: FM8501: A Verified Microprocessor. Springer-Verlag LNAI 795, Heidelberg (1994)
2. Manolios, P.: Mechanical Verification of Reactive Systems. PhD thesis, Department of Computer Sciences, The University of Texas at Austin (2001)
3. Aagaard, M.D., Cook, B., Day, N., Jones, R.B.: A Framework for Microprocessor Correctness Statements. In: Correct Hardware Design and Verification Methods (CHARME). Volume 2144 of LNCS., Springer-Verlag (2001) 443–448
4. Burch, J.R., Dill, D.L.: Automatic Verification of Pipelined Microprocessor Control. In Dill, D., ed.: Computer-Aided Verification (CAV). Volume 818 of LNCS., Springer-Verlag (1994) 68–80
5. Srivas, M., Bickford, M.: Formal Verification of a Pipelined Microprocessor. IEEE Software (1990) 52–64
6. Bronstein, A., Talcott, T.L.: Formal Verification of Pipelines based on String-functional Semantics. In Claesen, L.J.M., ed.: Formal VLSI Correctness Verification, VLSI Design Methods II. (1990) 349–366
7. Sawada, J., W. A. Hunt, Jr: Trace Table Based Approach for Pipelined Microprocessor Verification. In: Computer-Aided Verification (CAV). Volume 1254 of LNCS., Springer-Verlag (1997) 364–375
8. Sawada, J., W. A. Hunt, Jr: Processor Verification with Precise Exceptions and Speculative Execution. In Hu, A.J., Vardi, M.Y., eds.: Computer-Aided Verification (CAV). Volume 1427 of LNCS., Springer-Verlag (1998) 135–146
9. Hosabettu, R., Gopalakrishnan, G., Srivas, M.: Verifying Advanced Microarchitectures that Support Speculation and Exceptions. In: Computer-Aided Verification (CAV). Volume 1855 of LNCS., Springer-Verlag (2000)
10. Jhala, R., McMillan, K.: Microarchitecture Verification by Compositional Model Checking. In: Proceedings of Twelveth International Conference on Computer-aided Verification (CAV). Volume 2102 of LNCS., Springer-Verlag (2001)
11. Bryant, R.E., German, S., Velev, M.N.: Exploiting Positive Equality in a Logic of Equality with Uninterpreted Functions. In N. Halbwachs and D. Peled, ed.: Computer-Aided Verification (CAV). Volume 1633 of LNCS., Springer-Verlag (1999) 470–482
12. Lahiri, S.K., Bryant, R.E.: Deductive Verification of Advanced Out-of-Order Microprocessors. In W. A. Hunt, Jr, Somenzi, F., eds.: Computer-Aided Verification (CAV). Volume 2275 of LNCS., Springer-Verlag (2003) 341–354
13. Manolios, P.: Correctness of pipelined machines. In W. A. Hunt, Jr, Johnson, S.D., eds.: Third International Conference on Formal Methods in Computer-Aided Design (FMCAD). Volume 1954 of LNCS., Springer-Verlag (2000) 161–178
14. Manolios, P., Namjoshi, K., Sumners, R.: Linking Model-checking and Theorem-proving with Well-founded Bisimulations. In Halbwacha, N., Peled, D., eds.: Computer-Aided Verification (CAV). Volume 1633 of LNCS. (1999) 369–379
15. Sumners, R.: An Incremental Stuttering Refinement Proof of a Concurrent Program in ACL2. In Kaufmann, M., Moore, J.S., eds.: Second International Workshop on ACL2 Theorem Prover and Its Applications, Austin, TX (2000)
16. Kaufmann, M., Manolios, P., Moore, J.S.: Computer-Aided Reasoning: An Approach. Kluwer Academic Publishers (2000)
17. Kaufmann, M., Manolios, P., Moore, J.S., eds.: Computer-Aided Reasoning: ACL2 Case Studies. Kluwer Academic Publishers (2000)
18. Moore, J.S.: Proving Theorems about Java and the JVM with ACL2. Models, Algebras, and Logic of Engineering Software (2003) 227–290